OpenCSG: A Library for Image-Based CSG Rendering

Florian Kirsch, Jürgen Döllner

University of Potsdam, Hasso-Plattner-Institute, Germany
{florian.kirsch, juergen.doellner}@hpi.uni-potsdam.de

Abstract

We present the design and implementation of a real-time 3D graphics library for image-based Constructive Solid Geometry (CSG). This major approach of 3D modeling has not been supported by real-time computer graphics until recently. We explain two essential image-based CSG rendering algorithms, and we introduce an API that provides a compact access to their complex functionality and implementation. As an important feature, the CSG library seamlessly integrates application-defined 3D shapes as primitives of CSG operations to ensure high adaptability and openness. We also outline optimization techniques to improve the performance in the case of complex CSG models. A number of use cases demonstrate potential applications of the library.

1 Introduction

Constructive Solid Geometry (CSG) represents a powerful 3D modeling technique. The idea of CSG is to combine simple 3D shapes to more complex ones with Boolean operations in 3-dimensional space. Even though CSG is an established technique and it is well-understood, it has not become mainstream in software systems due to the complex implementations of rendering algorithms to display CSG models.

In the scope of CSG, the most basic shapes are called primitives. A CSG primitive must be solid, i.e., given in a way that interior and exterior regions of the primitive are clearly defined. For example, a sphere or a cube is a solid primitive, whereas a triangle is not. Two primitives or CSG shapes can be combined by one of the following Boolean operations to define a more complex CSG shape:

- **Union**. The resulting shape consists of all regions either in the first, in the second, or in both input shapes.
- **Intersection**. The resulting shape is the region common to both input shapes.
- **Subtraction**. The resulting shape is the region of the first shape, reduced by the region of the second shape.

CSG models are stored in CSG trees, where leaf nodes contain primitives and inner nodes contain Boolean operations (Figure 1). Because of the mathematical properties of Boolean operations, the resulting CSG shapes are always solid. This is an important advantage over other 3D modeling techniques, which often miss polygons and generate unclosed models.

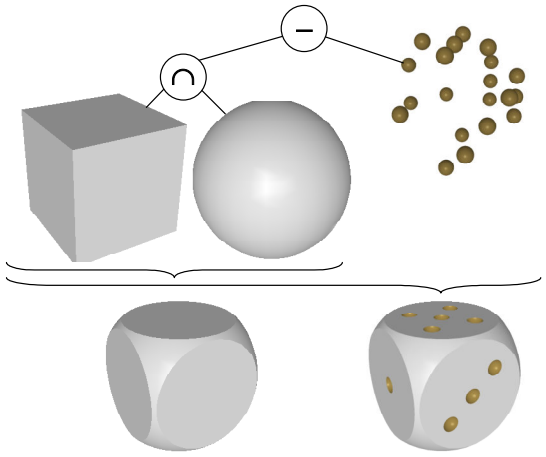

Figure 1. Modeling a dice using CSG. A cube and a sphere are intersected; from the result the dots of the dice are subtracted.

CSG modeling and rendering is directly available in several graphics systems for offline rendering, such as POV-Ray [1] or RenderMan [2]. Those graphics systems are used to generate photo-realistic images, and they are not suited for real-time rendering, though.

Rendering CSG shapes in real-time using and taking advantage of graphics hardware is difficult, in particular if the CSG shape is modified interactively. Basically, there are two options: On the one hand, the boundary of the CSG shape can be calculated mathematically and stored in a polygonal model that then is sent to graphics hardware. This is practical for static CSG shapes, but for CSG shapes that are modified interactively, the expensive calculation of the boundary must be repeated for each rendering frame, forbidding animated real-time display.

On the other hand, *image-based CSG rendering algorithms* can determine and store the visible parts of the CSG shape directly in the frame buffer of the graphics hardware. The result of an image-based CSG algorithm is, therefore, just the image of the CSG shape. Based on recent advances of graphics hardware images of CSG models can be generated instantaneously and, for models of considerable complexity, in real-time.

Image-based CSG can be used in all situations where complex 3D modeling operations are required in real-time. Image-based CSG does not analytically calculate the geometry of 3D objects, but most uses of CSG do not require the explicit 3D geometry and are satisfied with the image of the model. Still, image-based CSG is not commonly used in real-world applications today. Very often, the primitives of CSG shapes are only sketched as wire-frame image, which leads to difficult understanding of the final 3D-shapes' look.

We have developed OpenCSG, which represents the first free library for image-based CSG rendering. OpenCSG is written in C++, bases on OpenGL, and implements the two most important image-based CSG algorithms: The Goldfeather algorithm, which is suited for all kinds of CSG primitives, and the SCS algorithm, which is a more optimal algorithm if a CSG shape is composed of only convex CSG primitives.

2 Related Work

Constructive Solid Geometry (CSG) has been recognized as powerful approach for modeling complex 3D geometry for a long time [3]. The foundation of image-based CSG rendering was invented by Goldfeather et al. [4]. They described the normalization of CSG trees and also developed the first implementation of an image-based CSG algorithm. Another important class of CSG rendering algorithm today is the SCS algorithm [5].

2.1 Normalization of CSG Trees

Rendering arbitrary CSG trees directly in real-time is still not possible today. Instead, a CSG tree must first be normalized. A *normalized* CSG tree is in sum-of-products form, i.e., it is the union of several *CSG products* that consist, respectively, of a CSG tree with intersection and subtraction operations only, and only one single primitive is allowed to be the second operand of each operation. In other words, a CSG product has the form $(\ldots(x_1 \otimes x_2) \otimes x_3) \ldots \otimes x_n)$ where "$\otimes$" is either an intersection or a subtraction. Goldfeather et al. proved that the following set of equivalences, when applied repeatedly to inner nodes of an arbitrary CSG tree, transform the tree to a normalized CSG tree:

1. $x - (y \cup z) \rightarrow (x - y) - z$
2. $x \cap (y \cup z) \rightarrow (x \cap y) \cup (x \cap z)$
3. $x - (y \cap z) \rightarrow (x - y) \cup (x - z)$
4. $x \cap (y \cap z) \rightarrow (x \cap y) \cap z$
5. $x - (y - z) \rightarrow (x - y) \cup (x \cap z)$
6. $x \cap (y - z) \rightarrow (x \cap y) - z$
7. $(x - y) \cap z \rightarrow (x \cap z) - y$
8. $(x \cup y) - z \rightarrow (x - z) \cup (y - z)$
9. $(x \cup y) \cap z \rightarrow (x \cap z) \cup (y \cap z)$

Both the Goldfeather and the SCS algorithm render one CSG product at a time. The union of several products is determined by normal use of the depth buffer.

2.2 The Goldfeather Algorithm

The Goldfeather algorithm, for the case of convex primitives, bases on several observations (Figure 2): First, only the front face of intersected and the back face of subtracted primitives are potentially visible. Second, each other primitive *P* affects the visibility of a pixel in a potentially visible polygon by the number of polygons of *P* in front of the pixel (called the *parity*, which, for convex primitives, is always in-between zero and two): If *P* is subtracted and the parity is one, the pixel is not visible, and if *P* is intersected and the parity is zero or two, the pixel is also not visible. In all other cases, the pixel could be visible, as at least primitive *P* does not affect its visibility.

Based on these observations, the Goldfeather algorithm works as follows: It tests the visibility of each primitive in a CSG product separately. For this, the front respectively back face of the primitive is rendered into a temporary, empty depth buffer. Then, for all other primitives in the CSG product, the parity is determined, i.e., the number of polygons of the primitive in front of the depth buffer is counted. On modern graphics hardware, this is possible by toggling a bit in the stencil buffer [6], an additional kind of depth buffer that supports Boolean operations. Finally, if no parity

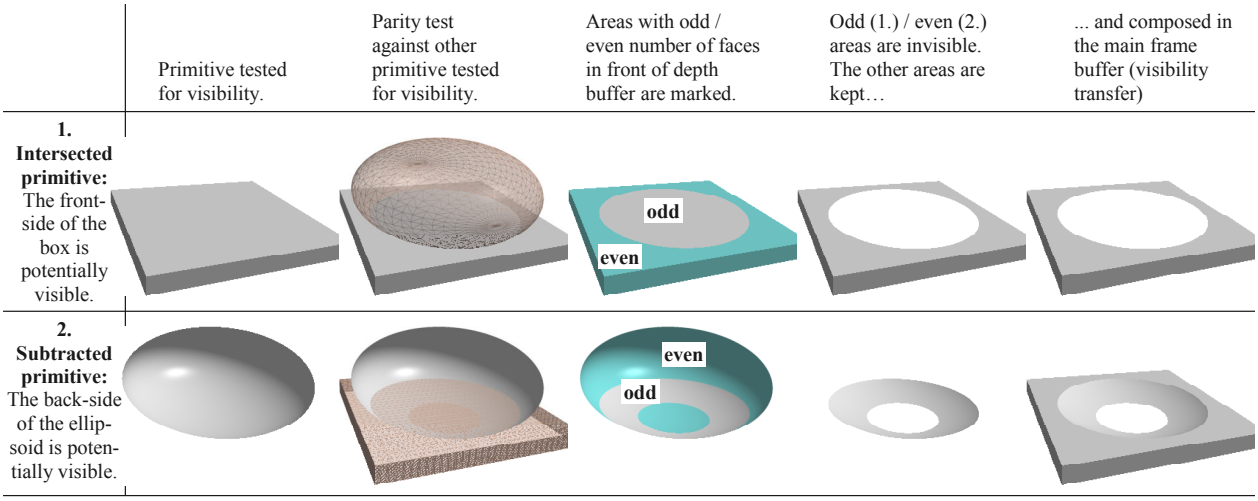

Figure 2. Example of the Goldfeather algorithm. An ellipsoid is subtracted from a box. The two primitives are tested for visibility separately. For each primitive, this requires calculating and analyzing the parity. When visible areas of a primitive have been determined, they are combined with the former content in the main frame buffer.

indicates that a pixel is invisible due to other primitives, the pixel is marked as visible, whereas the depth values of all other pixels are reset. The temporary depth buffer then is combined with the content of the main depth buffer before the algorithm continues with testing the visibility of the next primitive.

The basic Goldfeather algorithm can be used with a limited set of primitives, but some refinements are required to use the algorithm in more general cases:

Support of concave primitives [4]. First, the visibility of each potentially visible layer of a primitive must be checked separately. Layers of primitives can be rendered with the stencil test. Second, the parity calculation must be extended to distinguish an odd from an even number of polygons in front of a pixel. In practice, this is still possible by toggling a bit in the stencil buffer.

Resolution of the stencil buffer [6]. The stencil buffer typically only holds 8 bits for each pixel, such that it can only store the result of 8 parity tests at once. If more primitives are contained in a CSG product, invisible pixels must be remembered as such before continuing with the calculation of the remaining parities.

The following improvements and optimizations drastically increase rendering performance and, for complex CSG shapes, make image-based CSG rendering usable in practice:

Emulating two depth buffers. The Goldfeather algorithm requires two depth buffers, whereas graphics hardware directly exposes only one. This requires extensive copying between the depth buffer and main memory to emulate two depth buffers [6], or (far better) using an additional offscreen rendering canvas that holds the temporary depth buffer [7].

Visibility transfer. To combine the content of the temporary depth buffer with the main depth buffer, the depth values could be just copied with pixel operations [6]. But this requires a round-trip of the depth values to main memory. It is, therefore, faster by far to use a texture-based visibility transfer that can work completely on the graphics hardware [7].

Depth complexity. The basic Goldfeather algorithm has $O(n^2)$ run-time complexity, where n is the number of primitives in a CSG product. For large CSG products, it is often better to test the visibility of a complete depth layer at once [8]. This leads to a run-time complexity of $O(n \cdot k)$, where k is the depth complexity of the CSG product. The depth complexity can be calculated either using the stencil test or with occlusion queries [9].

Object-space optimizations. Primitives in a CSG product that do not overlap in screen space do not modify the visibility of one another. Therefore, the visibility of such primitives can be tested at once, and also the mutual calculation of the parity is not required (Section 6.3).

Image-space optimizations. Visible parts of a CSG product can only be inside the intersection of the bounding boxes of all intersected primitives in screen-space. The remaining areas do not need to be considered at all and can be omitted, for example, with the scissor test (Section 6.3).

2.3 The SCS Algorithm

The SCS Algorithm is optimized for convex primitives. In practical cases, it is typically faster than the Goldfeather algorithm, but in exchange it does not operate on concave primitives.

The SCS algorithm, similar to the Goldfeather algorithm, requires a normalized CSG tree. Its performance advantage is due to the determination of depth values of a complete CSG product in a temporary depth buffer at once. The algorithm has three stages: First, it determines the backmost front-face of all intersected primitives in the CSG product. Then, it subtracts primitives, i.e., where the front-face of a subtracted primitive would be visible and the back-face not, it replaces values in the depth buffer with the back-face depths of the subtracted primitive. For each subtracted primitive in a CSG product, this has to be done several times. Finally, the algorithm clips the depth buffer with the back-faces of all intersected primitives in the CSG product.

The SCS algorithm is subject to similar performance optimizations as the Goldfeather algorithm: It uses two depth buffers and requires a similar visibility transfer. It can also profit from knowing the depth complexity of a CSG product and object-space respectively image-space optimizations can be applied in a similar way.

2.4 Further Algorithms

Further image-based CSG algorithms appear to use two or more depth tests at the same time for rendering [10]. For a long time, standard graphics hardware did not support this. Guha et al. use the shadow mapping capability of modern graphics hardware for two-sided depth tests to implement their CSG algorithm [11]. This algorithm represents no fundamental new CSG rendering paradigm; It could be integrated into OpenCSG once hardware support matures.

2.5 Image-Based CSG Libraries

The only library for image-based CSG we are aware of is TGS SolidViz [12]. As this library is part of TGS OpenInventor 5.0, applications that are not based on OpenInventor, a rather large scene-graph library, cannot deploy SolidViz easily. TGS OpenInventor is available under a commercial license, the source code of SolidViz is not freely available, and the capabilities and restrictions of SolidViz are hardly documented.

Other implementations for real-time CSG rendering appear to be prove-of-concepts, which are demonstrated with small example programs. Therefore, they can be hardly integrated into other applications.

3 Overview of the OpenCSG Library

In the following, we describe our approach for the library for image-based CSG rendering. We will motivate the design choices in this section before we describe the consequences for implementation and usage of the library.

We consider the following points as important properties of a library for image-based CSG rendering:

1. A minimal, abstract, and well-defined interface, for easy use of the library. OpenCSG does not assume any specific type of CSG implementation.

2. A simple and well-defined output. The library is solely used for CSG rendering. All other tasks such as shading the CSG primitives are handled outside of the library.

3. Direct applicability to all kinds of rendering applications. Most graphics applications define their own set of graphical primitives, which are likely valid CSG primitives. It must be possible to use these graphical primitives for CSG rendering.

4. As few external dependencies as possible. A library that depends on a full-featured scene-graph library is clearly not acceptable.

5. Stability, performance, and portability as properties that every (graphics) library should provide.

OpenGL is the rendering library of choice to create portable real-time graphics applications that take advantage of the graphics hardware [13]; therefore, we use it for OpenCSG. Additionally, we require two small libraries: GLEW [14] is a library which manages loading and using OpenGL extensions, which provide a mean for using new rendering functionality that has not (yet) been adopted by the core OpenGL library. RenderTexture [15] is another small library that provides platform-independent access for hardware-accelerated offscreen rendering into textures by the means of p-buffers [16], a rendering technique that is extensively used by OpenCSG. Both GLEW and RenderTexture are, currently, statically linked with OpenCSG.

The most portable programming language for implementing a rendering library is C. However, the API of the library can be greatly simplified by using an object-oriented language such as C++ instead of C, especially for supporting user-defined primitives (design goal 3): In C, the rendering function of a CSG primitive would be specified using a function pointer. C++, as language with polymorphic objects, allows the same in a more convenient way by declaring an abstract rendering method in a primitive base class and implementing this method in derived classes. For this reason, OpenCSG is implemented in C++ instead of C.

4 The API of OpenCSG

In this section, we shortly describe the complete API of OpenCSG. The API is very compact and consists of only one class for CSG primitives and a rendering function that has a list of CSG primitives as argument and renders the CSG product indicated by this list of primitives into the depth buffer.

All classes, functions, and enumerations of OpenCSG are members of the C++ namespace `OpenCSG`.

4.1 Specifying Primitives

The interface of OpenCSG defines an abstract base class for all kinds of CSG primitives. Primitive objects can be assigned a bounding box in normalized device coordinates (screen-space mapped to $[-1, 1]^2$), which is used for object-based and image-based rendering optimizations internally. Setting the bounding box is optional. Primitives also have a convexity, i.e., the maximum number of depth layers of the primitive. The developer must set the convexity because some algorithms such as the Goldfeather algorithm need to know the convexity of a primitive for correct rendering operation. The convexity also allows for choosing between different CSG rendering algorithms that are provided by OpenCSG

The render method of the CSG primitive class is abstract. To use OpenCSG the developer must derive a specialized primitive class, which implements the render method. The CSG rendering function requires that the geometric transformation used for rendering a CSG primitive does not depend on transformations of other primitives. This is best done by pushing the transformation matrix at the beginning of the primitive's render function, and restoring the matrix from the transformation stack at the end. An alternative approach is to load the correct transformation matrix unconditionally for all primitives in a CSG product at the beginning of the primitive's render functions.

For internal use in OpenCSG, the render method of a primitive must not change the primary color. The color of a primitive is used by all CSG rendering algorithms for the texture-based visibility transfer, hence altering it in the rendering method causes invalid rendering results. For best rendering performance vertex positions alone should be submitted to graphics hardware because only pure geometry is needed for correct operation of CSG rendering. All other per-vertex data such as normals or texture coordinates is ignored.

```
namespace OpenCSG {
    enum Operation { Intersection, Subtraction };
    class Primitive {
    public:
        Primitive(Operation, unsigned int convexity);
        virtual ~Primitive();
        void setOperation(Operation);
        Operation getOperation() const;
        void setConvexity(unsigned int);
        unsigned int getConvexity() const;
        void setBoundingBox(float  minx, float  miny, float  minz,
                            float  maxx, float  maxy, float  maxz);
        void getBoundingBox(float& minx, float& miny, float& minz,
                            float& maxx, float& maxy, float& maxz) const;
        virtual void render() = 0;
    };
    enum Algorithm {
        Automatic, Goldfeather, SCS
    };
    enum DepthComplexityAlgorithm {
        NoDepthComplexitySampling, OcclusionQuery, DepthComplexitySampling
    };
    void render(const std::vector<Primitive*>& primitives,
                Algorithm = Automatic,
                DepthComplexityAlgorithm = NoDepthComplexitySampling);
}
```

Figure 3. The API of OpenCSG

4.2 Rendering CSG Models

The central part of the OpenCSG library are the CSG rendering algorithms. For using them, the API of OpenCSG defines a rendering function, which requires an argument containing the CSG shape to render. During the design of the API we considered two different options for that argument: It could be a full-featured CSG tree or only a CSG product, i.e., a list of primitives, each of them either subtracted or intersected. In the following, we discuss these two approaches.

First consider the required steps for the user of OpenCSG if the argument of the CSG rendering function would be a CSG tree. In this case the application developer would have to compose the CSG tree in its application. This would require a class for CSG primitives and some node classes for the inner nodes of the CSG tree, i.e., union, intersection, and subtraction. While this would not be particularly difficult, it would nonetheless require a complete API for composing and modifying CSG trees.

The second option is to permit only a CSG product as argument of the rendering function. In this case the interface can be drastically reduced: The rendering function only requires a list of CSG primitives that can be just given as STL `vector`. Additionally each CSG primitive must hold a flag to indicate whether it is subtracted or intersected.

If the argument of the CSG rendering function is a complete CSG tree, the rendering library must also implement the normalization of the CSG tree into a set of CSG products. In the other case, this normalization step has to be potentially implemented by the application developer. But implementing the normalization itself is not particularly difficult. Also, an application can often create a normalized CSG tree of smaller size than a general function in a CSG library because it can additionally analyze application-specific information. Furthermore, if the rendering function supported an arbitrary CSG tree as argument, the application might nonetheless require a non-trivial transformation of its CSG data to the CSG tree argument of the rendering function, depending on the kind of Boolean operations supported by the application.

Therefore, the render function of OpenCSG takes a CSG product as argument, given as a list of CSG primitives. We assume general functionality for normalizing a CSG tree would be better implemented by a separate library.

Furthermore, the render function has two optional arguments to choose between different CSG algorithms. The first of these arguments chooses between the Goldfeather algorithm, the SCS algorithm, or an automatic mode that chooses between both algorithms depending on the primitives in the CSG product. The second of these arguments chooses between different strategies for analyzing the depth complexity of the CSG product: The depth complexity can be ignored, determined directly by counting depth layers in the stencil buffer, or used indirectly with occlusion queries.

5 Output Generated by OpenCSG

The CSG rendering algorithms in OpenCSG work all in a similar way. From perspective of their rendering result they are exchangeable: They initialize the depth information in the depth buffer with the correct depth values of the CSG product. The depth values are written with respect to the depth values that were in the depth buffer before the CSG rendering call, using the common z-less depth function. This allows for scene composition of different CSG products with correct hidden surface removal.

Besides altering the depth buffer, the CSG rendering algorithms try to avoid any side effect. For example, OpenCSG does not alter the color buffer, i.e., it does not shade the colors of CSG shapes. This is because of the many different approaches and techniques for color shading. A number of graphics libraries are concerned with the task to do color shading well. There are even algorithms such as shadow mapping that do not require color shading of CSG primitives at all. Therefore, shading the primitives after initializing the depth buffer is better handled by the application.

The stencil buffer as remaining important part of the frame buffer should also not be affected by using the CSG rendering function. Currently, this is a technical problem for CSG shapes that contain concave primitives. During the visibility transfer, the CSG rendering function must render separate depth layers of these primitives in the frame buffer. This functionality is implemented by counting the depth layers in the stencil buffer, thereby destroying its former content. Hence, the CSG rendering function guaranties to preserve the stencil buffer only in the case of convex primitives, otherwise the content of the stencil buffer is undefined.

The CSG rendering function respects a number of settings of the OpenGL state for CSG rendering, whereas it ignores other settings that are required internally or simply do not make sense: Above all, the numerous OpenGL settings that manipulate color calculation are irrelevant for the CSG rendering function,

which does not alter the color buffer. The meaning of other important OpenGL settings for CSG rendering with OpenCSG is summarized in the following.

Alpha Test. OpenCSG internally uses the alpha test for visibility transfer. Therefore the application settings of the alpha test are ignored.

Scissor Test. This fragment test is applied as usual, i.e., fragments are only generated inside of the scissoring area.

Stencil Test. The stencil test will be performed correctly if only convex primitives are contained in the CSG product and if no layered CSG algorithms is used. In the other cases, our algorithms uses the stencil test internally and also overwrite the stencil buffer of the frame buffer.

Depth Test. Depth-test settings are completely managed by OpenCSG. It renders with the z-less depth function, i.e., the forefront of the CSG shape is rendered where it is in front of the former depth value in the depth buffer. The z-greater function would be the only potential alternative: The effect would be to render the backside of the CSG shape behind the former depth value in the depth buffer. Since the benefit of this setting is questionable and since it requires large changes to the implementation of the internal CSG algorithm, we did not implement this.

Culling. The setting of front- and back-face culling is ignored because OpenCSG uses this setting internally. Anyway, culling also lacks a sound meaning for CSG rendering of possibly concave CSG shapes.

Geometric Transformations. OpenCSG does not change the transformation and projection internally. It is under the responsibility of the CSG primitives to set and restore the necessary transformations.

6 Implementation Details

The implementation of the CSG algorithms in OpenCSG is similar to the algorithms presented by Kirsch and Döllner [7], and it uses many of the techniques presented there to improve rendering performance. Additionally, OpenCSG performs both object-based and image-based performance optimizations.

6.1 Choosing the CSG Algorithm

The CSG rendering function for rendering a CSG product has two parameters that control what kind of algorithm is used for CSG rendering. As basic algorithms, the Goldfeather algorithm and the SCS algorithm are provided. Each of them can by used in several variants: The depth complexity of the CSG product can be determined (1) by counting the overdraw of the CSG product in the stencil buffer, (2) indirectly by the means of occlusion queries, or (3) not at all.

OpenCSG users can also specify the automatic mode, in which an internal heuristic guesses the fastest algorithm for rendering the CSG product. The following heuristic experimentally turned out to give satisfactory results: The SCS algorithm is chosen if the CSG product contains only convex primitives, else the Goldfeather algorithm is used. As strategy for depth complexity, hardware occlusion queries are used if the CSG product contains more than 20 primitives and the hardware supports them. If the hardware does not support them and the CSG product contains even more than 40 primitives, depth complexity is calculated by counting the overdraw of the CSG product with the stencil buffer. In all other cases, the depth complexity is not determined at all.

6.2 P-Buffer Settings

For supporting two depth buffers, OpenCSG uses p-buffers, i.e., offscreen rendering canvases. The visibility transfer from temporary to main depth buffer is performed using RGBA-textures, i.e., a texture is created in which each color channel encodes the visibility of a CSG primitive (Goldfeather) respectively the alpha channel encodes one CSG product (SCS). This texture is used to reconstruct the depth values in the main depth buffer.

The p-buffer for internal calculation of the primitive's visibility must be configured carefully such that it behaves identically to the main frame-buffer. This is required because p-buffers have their own rendering context with rendering settings that are independent of the main rendering context. So we replicate the viewport-size, the transformation and projection matrix, and the front-face setting.

Determining an optimal physical size of the p-buffer is not trivial. First we tried the approach to set the size of the p-buffer to the size of the frame buffer in the calling application (or to the next power-of-two size in case of missing support for non-power-of-two-textures). But this approach does not work well for applications using two or more frame buffers displaying CSG shapes. As the sizes of the frame buffers likely differ the p-buffer is constantly resized resulting in unacceptable rendering performance. This does not only

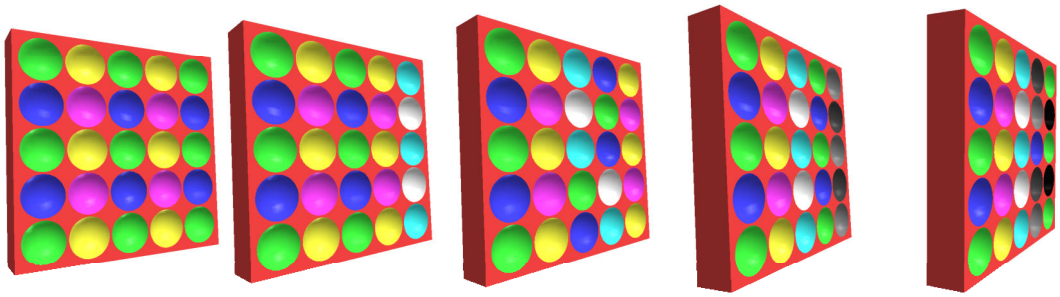

Figure 4: Primitive Batches. In these images, primitives that are contained in a batch are depicted with the same color. The number of primitive batches depends of the orientation of the CSG product with respect to the camera: In the left image, there are six primitive batches (included the box); in the right image there are twelve.

occur in applications with several OpenGL-windows; it is also an issue for multipass algorithms such as dynamic shadow mapping or reflections by the means of textures because those algorithms internally require additional frame buffers.

We solved this problem with a heuristic for resizing the p-buffer consisting of two rules: If the size of the current p-buffer is smaller than the requested size x_0 x y_0, the p-buffer is enlarged to that size immediately. Otherwise we analyze the requested sizes in both dimensions during the last N requests $x_0 \ldots x_N$ and $y_0 \ldots y_N$ and calculate the respective maxima $x_{max} = \max \{x_i \mid i \in 0\ldots N\}$ and $y_{max} = \max \{y_i \mid i \in 0\ldots N\}$. If for a request the size of the current p-buffer exceeds x_{max} x y_{max}, we downsize the p-buffer to the new size x_{max} x y_{max}. In practice, a value of 100 for N proves to get good results: The p-buffer is always big enough, it is not resized too often, and memory for an unnecessary large p-buffer is not occupied forever.

6.3 Performance Optimizations

Technical publications about image-based CSG rarely mention the fact that the performance of CSG rendering can be drastically improved with rather simple optimizations that just take into account the bounding boxes of primitives in the CSG product. OpenCSG implements the following optimizations:

Primitive Batches. For the standard Goldfeather algorithm, normally the visibility of each primitive is calculated separately. But obviously, primitives that do not overlap in z-direction of the camera space do not mutually influence themselves. Therefore, primitives whose bounding boxes do not overlap in z-direction are internally grouped to batches of primitives (Figure 4). Further on, each batch is treated as a single primitive. This way we reduce the number of parity tests and also lower the number of copy operations from the temporary into the main depth buffer. The same idea is applied to the SCS algorithm with all depth-complexity strategies, whereas it makes no sense for the Goldfeather algorithm using depth-complexity sampling.

Disjoint Bounding Volumes. In the standard Goldfeather algorithm, calculating the parity of a subtracted primitive against a primitive batch is only required if its bounding box intersects at least one bounding box of a primitive in the primitive batch. Otherwise, the subtracted primitive cannot alter the visibility of the primitive batch anyway. This optimization does neither apply to the Goldfeather algorithm using depth-complexity sampling nor to the SCS algorithm.

Restrict Rendering to Intersection Area. Visible parts of the CSG product can only be found where intersected primitives overlap in the xy-plane of camera space. Hence, we calculate the bounding boxes of all intersected primitives in pixel coordinates, intersect them, and restrict rendering during the complete calculation of the CSG product to this *intersection area* using scissoring. This optimization is applicable to all image-based CSG algorithms.

Restrict Rendering to Primitive Batch. For the parity calculation in the standard Goldfeather algorithm, the scissoring area can be minimized even more: The parity does not need to be calculated for pixels outside of the bounding box of the primitive batch whose visibility is being determined because there are no visible pixels of the primitive batch anyway. Therefore, we set the scissor region such that the parity is only calculated in the intersection of the bounding box of the primitive batch and the aforementioned intersection area.

7 Using OpenCSG

In this section, we describe a small example for CSG rendering with OpenCSG in practice. We also give some practical hints for shading CSG primitives for more complex applications and lighting conditions.

7.1 Overview of the Rendering Process

Rendering CSG shapes with OpenCSG requires two steps: First, OpenCSG initializes the depth buffer of one or several CSG products. Then, the application is responsible for the color shading of the CSG primitives. For that, the primitives must be rendered using a z-equal depth function and with exactly the same transformations that are used in the CSG primitive objects. As only the front faces of intersected and the back faces of subtracted primitives can be visible, back or front face culling can be used accordingly to enhance rendering performance. The resulting performance improvements are, however, hardly noticeable. Culling is not important to get a correct rendering result, since the depth test for equality already filters out all invisible fragments.

7.2 A First Example

The OpenCSG library contains the example described in the following, which illustrates a simple but effective way to use it. The application creates OpenGL display lists for each CSG primitive, which are created using the OpenGL helper libraries GLU and GLUT.

```
GLuint id1 = glGenLists(1);
glNewList(id1, GL_COMPILE);
glutSolidCube(1.8);
glEndList();
GLuint id2 = glGenLists(1);
glNewList(id2, GL_COMPILE);
glutSolidSphere(1.2, 20, 20);
glEndList();
```

The application also derives a concrete class `DLPrim` for CSG primitives, whose render-method just executes one display-list id.

```
class DLPrim : public OpenCSG::Primitive {
public:
  DLPrim(unsigned int displayListId,
         OpenCSG::Operation operation,
         unsigned int convexity)
    : OpenCSG::Primitive(operation, convexity),
      id(displayListId) { }
  virtual void render() { glCallList(id); }
private:
  unsigned int id;
};
```

For each CSG primitive, an object of this class is created that holds the appropriate display list id for rendering. A CSG product as argument for the render function of OpenCSG is constructed just by appending several CSG primitives to an STL `vector`:

```
namespace OpenCSG;
DLPrim* box=new DLPrim(id1, Intersection, 1);
DLPrim* sphere=new DLPrim(id2,Subtraction, 1);
std::vector<Primitive*> primitives;
```

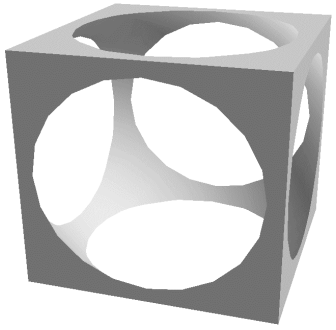

Figure 5. Resulting image of the CSG model defined in the example.

```
primitives.push_back(box);
primitives.push_back(sphere);
```

For rendering the CSG product described in this way, we invoke the render function with the list of primitives. Optionally, we can choose a specific algorithm and a strategy for depth complexity sampling.

```
OpenCSG::render(primitives,
                Goldfeather,
                NoDepthComplexitySampling);
```

After this, the depth buffer contains the correct depth values of the CSG product. Now the application is responsible for shading the CSG product with the desired colors. We set the depth comparison function to z-equal, such that only fragments are rendered that equal in depth to the value stored in the depth buffer. We also set up lighting and shading as desired and then render all primitives in the CSG product.

```
glDepthFunc(GL_EQUAL);
// setup lighting and shading
for (int i=0; i<primitives.size(); ++i) {
    (primitives[i])->render();
}
glDepthFunc(GL_LESS);
```

Finally, we reset depth testing to its original value. Figure 5 shows the image that is rendered by this example.

The bounding boxes of the primitives are not set in this example. This is fine for such a small CSG product with few optimization opportunities. But for CSG products that consist of more primitives setting the bounding boxes is likely to increase rendering performance.

7.3 Lighting CSG Shapes

For illuminating CSG shapes, it is important to distinguish between intersected and subtracted primitives: polygons of intersected primitives are oriented normally, but subtracted primitives have an inverse orientation because only their inner back-facing polygons can be visible. Therefore the lighting of subtracted

primitives is, without precautions, inverse to the standard lighting: polygons that point away from the light are lighted, but polygons that are oriented towards the light source are unlighted.

The first option to fix this issue is to negate the normals of all subtracted primitives. This is likely the best option if vertex programs are used for geometrical transformations because a vertex program can implement the negation of normals easily. Otherwise this is a potentially more complicated operation, since the normals of primitives must be created analytically, respecting the CSG operation of the primitive. Negating normals is the only option that works correctly for shading by the means of per-pixel lighting respectively bump-mapping in a fragment program.

The second option for the standard vertex lighting is to enable two-sided lighting. In this case, vertices of back-facing polygons have their normals reversed before the lighting equation is evaluated. Unfortunately, some graphics hardware such as the GeForceFX takes a big performance hit with two-sided lighting. Apart from this performance weakness, this is a simple solution for correct lighting.

A third workaround for the lighting problem is to place two identical light sources on the opposite sides of the CSG model: one of them illuminates polygons of intersected and the other one polygons of subtracted primitives facing to the same direction. While this may be a kludge, in practice this approach works well if we can accept an illumination that is equal from both sides of the CSG model.

8 Limitations and Performance

As a general problem of image-based CSG rendering, CSG shapes are only correctly rendered if all primitives are completely inside the view frustum. Otherwise the internal calculations of the CSG algorithms, for example calculating the parity, fail. The fundamental problem is that image-based CSG rendering requires solid primitives, but if a solid primitive crosses the front- or back plane of the view frustum, some parts are clipped and the primitive is not closed, and, therefore, not solid in the view frustum anymore.

With the SCS algorithm, an alpha-texture encodes the visibility of all primitives in a CSG product. As the alpha channel typically has a resolution of 8 bits only, 256 different values can be distinguished in a channel, therefore the CSG products may only consist of at most

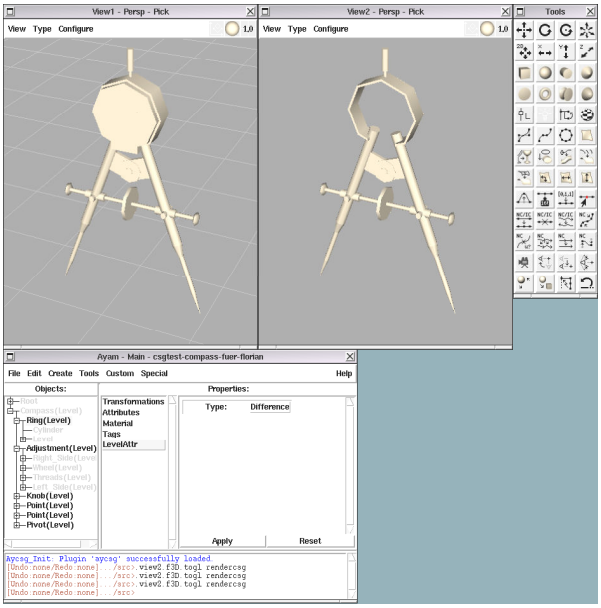

Figure 6. Screenshot of the RenderMan modeling environment Ayam using OpenCSG for previewing CSG models.

255 primitive batches. In our experience, this limitation can only be exceeded in pathological cases. As a future work, IDs could be spread over the whole RGBA channel, allowing 2^{32}–1 different primitive batches for a CSG product.

CSG rendering with OpenCSG requires some OpenGL extensions. At least, a p-buffer must be available for offscreen rendering. Also, a stencil buffer must be supported in the p-buffer, and, for rendering concave shapes, also in the main frame-buffer. However, very old graphic hardware does not support stencil or p- buffers. In practice, OpenCSG can be used on all newer graphics hardware: The oldest NVidia graphics chip that is supported by OpenCSG is the Riva TNT, and the oldest ATI generation is probably the Rage 128 (the latter hardware was, till now, not available for testing, though).

For acceptable performance, using more recent graphics hardware is important, whereas the power of the CPU does not matter much. We have observed excellent performance on GPUs such as the GeForce4, GeForceFX, or ATI Radeon 9800. On such graphics hardware, OpenCSG can use additional hardware features such as hardware occlusion queries for depth-complexity sampling and dot products in the texture environment, which are used to improve the performance of the visibility transfer.

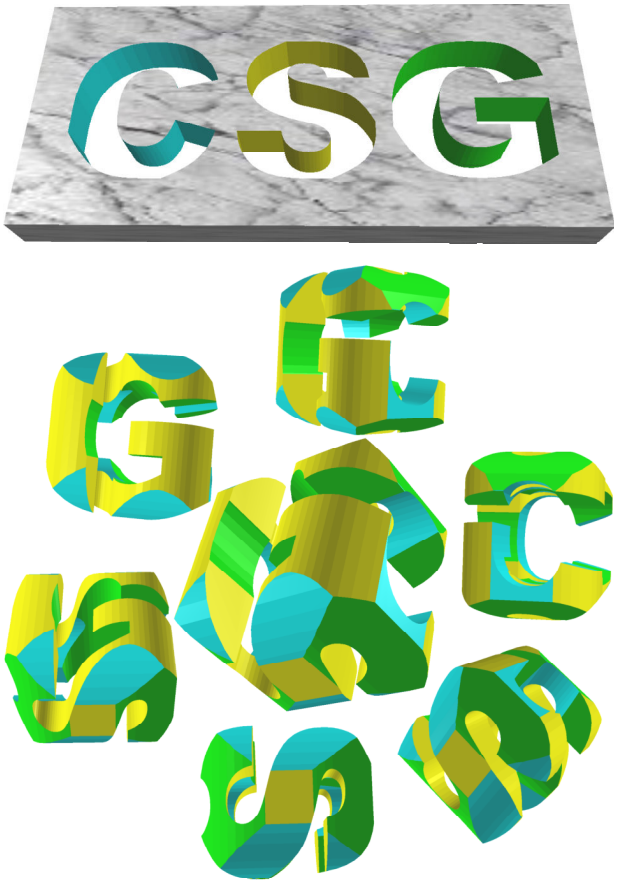

Figure 7. How does a 3D object look like that exactly fits through all letter-shaped holes? CSG gives the answer: Just intersect the extruded 3D-models of each letter. The resulting shape is shown below from different positions.

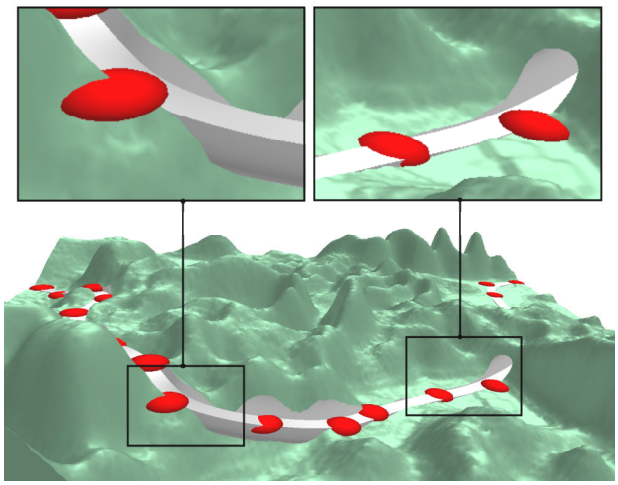

Figure 8. Interactive road design in a 3D terrain model using CSG operations to model excavations, embankments, and tunnels.

In practice, with an ATI 9800 and a window resolution of 800×600, the trivial examples in Figure 2, 4, and 5 achieve several hundred fps. The dice in Figure 1 renders at about 100 fps, the puzzle of Figure 7 at about 60 fps, and the terrain in Figure 8, which bases on a height-field of 256 x 256 resolution, at 22 fps. The terrain is a rather difficult CSG model because the terrain primitive is not convex and must be rendered with up to eight depth layers.

9 Other Applications

CSG modeling is common practice in offline-rendering systems, whereas interactive modeling environments for such rendering systems have not supported real-time CSG rendering traditionally. With OpenCSG, this is hopefully about to change. Ayam [17], the Tcl/Tk-based modeler for RenderMan compliant renderers, has been the first such application we are aware of that has integrated CSG rendering with OpenCSG in its most recent version (see Figure 6). Currently, the Blender [18] development community discusses integration of an OpenCSG-based rendering plug-in for real-time preview of CSG models.

With the availability of an open and stable CSG library, we foresee many novel uses of CSG. We have implemented some examples based on an OpenCSG plug-in for the scene graph library VRS [19]. One application for CSG can be found in the entertainment field, for example in puzzle games such as the jigsaw puzzle for the letters “C”, “S”, and “G” in Figure 7: A shape that fits exactly through the three letter-shaped holes is easily constructed by intersecting the extruded 3D-models of each letter.

Another new idea for using CSG beyond the scope of traditional modeling applications is road design in 3D terrain models (Figure 8). The red markers, which can be moved to all six directions interactively, define the course of the road. Thereby two tube-formed CSG primitives S_1 and S_2 compose the 3D-model of the road. The upper tube S_1 is subtracted from the terrain primitive T such that the final CSG expression is $(T - S_1) \cup S_2$. In this way, excavations, embankments, and even tunnels can be manipulated interactively and anticipated at an early design stage.

10 Conclusions

We have shown concepts and implementation of OpenCSG, an open graphics library for image-based rendering of CSG models. Our work demonstrates that

image-based CSG becomes a mature enabling technology for novel real-time 3D applications. In particular, we foresee applications in the fields of simulation, 3D modeling, CAD/CAM, and gaming.

OpenCSG also shows how a compact abstract API can provide access to a broad range of CSG algorithm variants with a minimum programming effort from an application developer's point-of-view: Due to its simple API, OpenCSG can be easily integrated into any OpenGL-based application. Provided the application-defined 3D shapes are solid, they can be directly processed by OpenCSG as CSG primitives.

The OpenCSG library has been developed since mid-2002 with the first public version released in September 2003 and version 1.0 in May 2004. It is provided under the GPL-license at `http://www.opencsg.org`.

Acknowledgements

We would like to thank Randolf Schultz for providing us with a high-quality screenshot of Ayam.

References

[1] *POV-Ray, The Persistence of Vision Raytracer*, `http://www.povray.org`

[2] S. Upstill, *The Renderman Companion*, Addison Wesley, 1989, ISBN 0201508680

[3] A. A. G. Requicha, *Representations for Rigid Solids: Theory, Methods, and Systems*, ACM Computing Surveys, 12(4):437-464, 1980.

[4] J. Goldfeather, S. Molnar, G. Turk, and H. Fuchs, *Near Realtime CSG Rendering Using Tree Normalization and Geometric Pruning*, IEEE Computer Graphics and Applications, 9(3):20-28, 1989.

[5] N. Stewart, G. Leach, and S. John, *Linear-time CSG Rendering of Intersected Convex Objects*, Journal of WSCG, 10(2):437-444, 2002.

[6] T. F. Wiegand, *Interactive Rendering of CSG Models*, Computer Graphics Forum, 15(4):249-261, 1996.

[7] F. Kirsch, and J. Döllner, *Rendering Techniques for Hardware-Accelerated Image-Based CSG*, Journal of WSCG, 12(2):221-228, 2004.

[8] N. Stewart, G. Leach, and S. John, *An Improved Z-Buffer CSG Rendering Algorithm*, 1998 Eurographics / SIGGRAPH Workshop on Graphics Hardware, ACM, 25-30, 1998.

[9] M. J. Kilgard, (editor), *NVIDIA OpenGL Extension Specifications*, April 2004. `http://developer.nvidia.com`

[10] D. Epstein, F. Jansen, and J. Rossignac, *Z-Buffer Rendering from CSG: The Trickle Algorithm*, IBM Research Report RC 15182, 1989.

[11] S. Guha, S. Krishnan, K. Munagala, and S. Venkatasubramanian, *Application of the Two-Sided Depth Test to CSG Rendering*, ACM SIGGRAPH 2003 Symposium on Interactive 3D Graphics, 177-180, 2003.

[12] *TGS SolidViz 1.2*, part of TGS OpenInventor 5.0 from Mercury, 2004, `http://www.tgs.com/support/oiv_doc`

[13] M. Segal, K. Akeley, *The OpenGL Graphics System: A Specification*, 2004, `http://www.opengl.org/documentation/spec.html`

[14] *GLEW: OpenGL Extension Wrangler Library*, `http://glew.sourceforge.net`

[15] *RenderTexture 2.0*, `http://gpgpu.sourceforge.net`

[16] C. Wynn, *OpenGL Render-to-Texture*, Presentation, NVidia Corporation, 2002, `http://developer.nvidia.com`

[17] *Ayam*, `http://ayam.sourceforge.net`

[18] *Blender*, `http://www.blender3d.com`

[19] *VRS – The Virtual Rendering System*, `http://www.vrs3d.org`